

# Cache-Conscious Run-time Decomposition of Data Parallel Computations

Hervé Paulino, Nuno Delgado

*NOVA Laboratory for Computer Science and Informatics & Departamento de Informática,  
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica,  
Portugal*

---

## Abstract

Multi-core architectures feature an intricate hierarchy of cache memories, with multiple levels and sizes. To adequately decompose an application according to the traits of a particular memory hierarchy is a cumbersome task that may be rewarded with significant performance gains. The current state-of-the-art in memory-hierarchy-aware parallel computing delegates this endeavour on the programmer, demanding from him deep knowledge of both parallel programming and computer architecture. In this paper we propose the shifting of these memory-hierarchy-related concerns to the run-time system, which then takes on the responsibility of distributing the computation's data across the target memory hierarchy. We evaluate our approach from a performance perspective, comparing it against the common cache-neglectful data decomposition strategy.

*Keywords:* Data parallelism, Data locality, Cache Optimizations, Parallel computing, Run-time support

---

## 1. Introduction

Data parallelism is the most common parallel decomposition strategy, by which an application's domain is decomposed into as many partitions as workers assigned to the computation. Such strategy is cache hierarchy neglectful and hence, in many cases, does not harvest the benefits provided by the (consistently growing amount of) cache hardware available in current computers, from laptops to high-end server nodes.

An adequate mapping of a computation onto the underlying memory hierarchy is crucial to fully harness the computational power of modern architectures. However, cache memory management is completely transparent to user-level programming. Such responsibility typically falls upon the hardware infrastructure, whose function is only to guarantee that recently accessed data is closer

---

<sup>☆</sup>Submitted to Journal of Supercomputing

<sup>☆☆</sup>This work was partially funded by FCT-MEC in the framework of the PEst-OE/EEI/UI0527/2014 strategic project.

*Email address:* herve.paulino@fct.unl.pt (Hervé Paulino)

*URL:* <http://asc.di.fct.unl.pt/~herve> (Hervé Paulino)

to the computing unit(s) than the remainder, since it will likely be accessed again. Cache replacement algorithms are based upon heuristics, such as *least-recently-used*, that do not always serve the application’s best interest, given that they base their decision on historic information rather than on information about future accesses. This limitation has a major impact on the performance of temporal locality sensitive computations, such as stencil computations and computations over matrices.

To overcome this problem, it is up to the compiler, the run-time system or, ultimately, the programmer to implement efficient, application-specific, mappings that maximize the cache hit ratio. Cache-guided optimizations in mainstream compilers consist essentially of loop transformations [1] directed at sequential loops, which, in the context of parallel computing, may only be applied to the internal execution of tasks. In fact, the data locality issue in parallel computing has been mostly addressed at language level, via linguistic constructions for the explicit programming of the memory hierarchy [2, 3, 4, 5, 6, 7]. However, these place a heavy burden on the programmer, requiring in-depth knowledge of parallel programming and computer architecture. Moreover, they apply divide-and-conquer strategies that cannot be applied in frameworks that cleanly separate the problem decomposition stage from the execution stage, such as MapReduce [8]. The same reasoning may be applied to cache-oblivious algorithms [9], which oblige the programmer to design divide-and-conquer algorithms that do not depend on the specificities of a particular cache hierarchy.

In this work we are particularly interested in the aforementioned class of computations, where an initial decomposition stage generates a set of partitions, upon which a computation stage is subsequently applied in parallel. More precisely, we are interested on exploring how this decomposition stage may enhance the data locality properties of the ensuing computation, so that the latter may transparently benefit from a good mapping onto the underlying memory hierarchy. To carry out such enterprise we have been researching on how to address some key challenges, namely on how to deduce mappings for specific memory hierarchy configurations from the same source code, and on how to ensure performance portability across a wide range of configurations. Overall, the main contributions of this paper are twofold:

1. domain decomposition principles and algorithms that take into consideration the complex memory hierarchies of current computer architectures, and
2. a study that compares our cache-conscious approach against the classical, cache-neglectful, strategy.

To the best of our knowledge, no such comparative study exists. Works such as [2, 3, 4] simply present speed-up analyses against sequential versions of the benchmarks. There is no evidence that the effort required from the programmer (to explicitly map the computation onto the memory hierarchy) actually delivers performance gains when compared to simpler strategies. Conversely, our study attests the validity of our proposal, showing that it delivers significant speed-ups to computations that are particularly sensitive to temporal locality.

The remainder of this paper is structured as follows: Sections 2 and 3 present the principles and implementation of our cache-conscious decomposition of data-parallel computations; Section 4 evaluates our proposal from a performance per-

spective, with a particular focus on the comparison against the classical strategy for domain decomposition; Section 5 positions our approach relatively to the current state-of-the-art; and, finally, Section 6 presents our final conclusions and prospective future work.

## 2. Cache-Conscious Domain Decomposition

Domain decomposition in distributed memory environments is a two stage operation: initially, the domain is partitioned among the nodes that compose the distributed system, and secondly, within each node, it is further partitioned among the worker threads locally assigned to the computation. We argue that these two stages must be clearly decoupled, so that stage-specific optimizations may be devised. For instance, handling heterogeneity only makes sense at *cluster* level, while cache-awareness only makes sense at *node* level.

The focus of this paper is on how to explore domain decomposition (at node level) to enhance the locality properties of data parallel computations. To that end, we leverage the cache hardware available at each node and apply a cache-conscious strategy that takes into account the characteristics of some (or all) levels of the target memory hierarchy, and not only of the worker threads assigned to the computation (the standard *horizontal* approach). As a result, the number of resulting partitions will be a function of the target machine’s cache hierarchy.

Figures 1 and 2 highlight the differences between the horizontal decomposition approach and our cache-conscious proposal. In the latter case, the domain is decomposed in such a way that each partition fits - its size is a function of - a given target cache level (TCL). Additionally, the behavior of each worker assigned to the computation is modified so that it iteratively applies the user-defined computation upon a stream of partitions, rather than to a single one. The number of workers is preserved, the amount of work performed by each worker is also generally preserved<sup>1</sup>, but the granularity of the data upon which the user-defined computation is individually applied is (potentially) much smaller, thus enhancing locality.

To successfully apply our strategy we have to address two distinct challenges: i) how to *decompose* a computation’s domain so that such computation may make better use of the available cache hardware and ii) how to efficiently *schedule* the resulting tasks onto the available set of workers, while also leveraging data locality.

### 2.1. Decomposition

For the sake of generality, we allow a computation’s input data-set (our domain to decompose) to be built from multiple sub-domains, typically individual data-structures, each with its own decomposition strategy. For example, in the classic matrix multiplication, one can implement a single decomposition strategy that splits the 3 matrices involved, or decompose the 3 matrices individually

---

<sup>1</sup>The user-defined decomposition algorithm may generate irregular partitions, ultimately leading to an unbalanced work distribution among the workers. The issue is orthogonal to the cache-conscious decomposition, which in general does not increase nor decrease such unbalance.

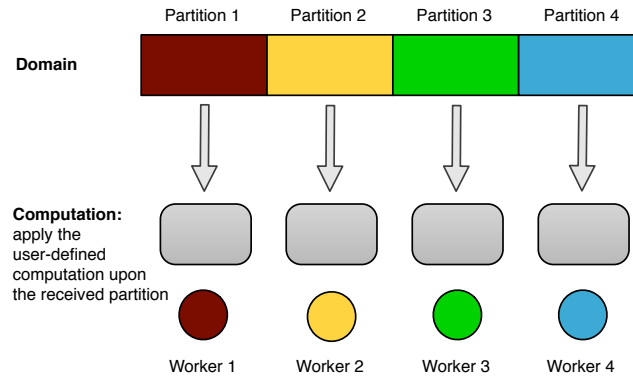


Figure 1: Application of a user-defined computation over a horizontally decomposed domain

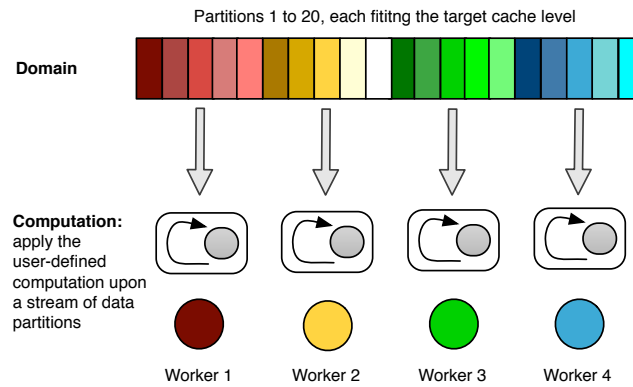


Figure 2: Application of a user-defined computation over a cache-consciously decomposed domain

|  |  |
|--|--|
| <code>partition(int np) : T[]</code>                 | Partitions the input domain into <code>np</code> partitions  |
| <code>validate(int np) : int</code>                  | Validates if the input domain may be partitioned into <code>np</code> partitions, result:<br>$< 0$ - there is no solution for any value $\geq np$ ,<br>$= 0$ - <code>np</code> is not a valid solution, but there may be solutions for values $> np$ ,<br>$> 0$ - <code>np</code> is a valid solution. |
| <code>getElementSize() : int</code>                  | Size of an element of <code>T</code> (in bytes)  |
| <code>getIndivisibleSize(int np) : int</code>        | Indivisible size of a partition (in number of elements)  |
| <code>getAveragePartitionSize(int np) : float</code> | Average size of a partition (in number of elements)  |
| <code>getAverageFirstDimSize(int np) : float</code>  | Average size of the first dimension of a partition (in number of elements)   |

Table 1: The `Distribution<T>` interface

and have a way of combining the resulting individual partitions. In this latter strategy, a partition of the input data-set must comprise a partition of each of its sub-domains.

Naturally, the number of indivisible units of each sub-domain may not be a multiple of the number of desired partitions. This fact may be easily solved by distributing the remainder units among the regular-sized partitions, causing an unbalancing of, at most, one indivisible unit. However, problem-specific constraints may impose further restrictions upon the number of partitions and/or the geometry of the decomposition as a whole. Stencil computations present this kind of restrictions with regard to the number of elements and their relative positions. For example, consider a computation over a 2-dimensional grid where, at instant  $t_{i+1}$ , the value of each of the grid's elements is a function of its own value and of its 8 adjacent elements at instant  $t_i$ . To meet these restrictions, each partition of the domain must comprise  $3 \times n$  elements, with  $n \geq 3$ , organized in grids of, at least, 3 rows  $\times$  3 columns.

This problem-specific information must be conveyed in the decomposition algorithms supplied by the programmer, that we refer to as *distributions*, and must therefore be included in the interface that regulates the implementation of such algorithms (Table 1). Additionally, in order to perform our cache-conscious systematic decomposition, we will require an extra set of functions, whose purpose we will unveil as we progress along this section.

#### 2.1.1. Determining the number and size of the partitions

Given a domain  $D$ , composed of  $d$  sub-domains ( $D = \bigcup_{i=0}^{d-1} D_i$ ), its decomposition into a set of partitions  $P_D$ , each fitting a given TCL, requires the calculation of a value for the number of partitions ( $np$ ) such that

$$\forall p \in P_D, \text{size}(TCL) \geq \text{size}(p) = \sum_{i=0}^{d-1} \frac{\text{size}(D_i)}{np}$$

where  $\text{size}()$  denotes the size in bytes of either the enclosed cache level or partition.

The proposed value of  $np$  must be validated by each of the distribution algorithms involved, being the verification logic defined in function `validate`. Algorithm 1 presents the procedure for assessing if a given number of partitions

---

**Input:** `TCL_PER_CORE` - Size in bytes of the TCL per core  
**Input:** `CACHE_LINE_SIZE` - Size in bytes of a cache coherence line  
**Input:** `nDomains` - Number of sub-domains that form the domain to decompose  
**Input:** `np` - Number of partitions into which each sub-domain must be decomposed  
**Input:** `dist`s - Vector holding the distribution algorithms for each sub-domain  
**Input:**  $\varphi$  - Function that estimates the size of a partition in cache  
**Output:** 1 - the value of `np` is valid  
**Output:** 0 - the value of `np` is not valid, but higher values may be  
**Output:** -1 - the value of `np` is not valid, nor are any higher values

```

1 totalPartitionSize  $\leftarrow$  0;
2 for  $i \leftarrow 0$  to nDomains do
3   status  $\leftarrow$  dists[ $i$ ].validate(np);
4   if status  $\leq$  0 then return status
   totalPartitionSize  $\leftarrow$  totalPartitionSize +  $\varphi$ (CACHE_LINE_SIZE, dists[ $i$ ], np);
5 end
6 if totalPartitionSize  $\leq$  TCL_PER_CORE then return 1 else return 0

```

---

**Algorithm 1:** Verification if a domain may be decomposed into a given number of partitions

(`np`) is valid: it assures that each sub-domain may be split into that many partitions, according to distribution algorithms involved (line 3), and that the cumulative size of such partitions (*totalPartSize*) fits in the TCL (line 7). The algorithm depends on an estimation of how many bytes a partition will occupy in the TCL. To enable the experimentation with different heuristics, the estimation is delegated on function  $\varphi$  (line 5), supplied as parameter. The outcome of the algorithm is the validation, or invalidation, of the candidate `np` value. In the later case, information concerning values higher than the candidate is also supplied. This information is used to delimit the search space, as subsequently explained.

To compute the optimal size of a partition that fits in a TCL, we apply a binary search: the value of `np` begins in the number of workers assigned to the execution (*nWorkers*) and doubles in every iteration until a valid solution is found or all values larger than `np` are invalid (according to Algorithm 1). From then onwards, the search's interval is continuously narrowed to find the smallest valid `np` value. Given that the size of each individual partition is inversely proportional to `np`, such solution is optimal for the provided input parameters. The *nWorkers* lower-bound guarantees that the algorithm generates, at least, as many partitions as available workers, in order to fully exploit the designated resources.

### 2.1.2. The $\varphi$ function

The definition of the  $\varphi$  function implies a trade-off between accuracy, computational overhead, and wasted cache space. A simple approach ( $\varphi_s$ ) is not to take into consideration either the size of the target architecture's cache line size (`CACHE_LINE_SIZE` in the algorithm) nor the partition's geometry. Thus, the function simply computes the number of bytes the partition takes:

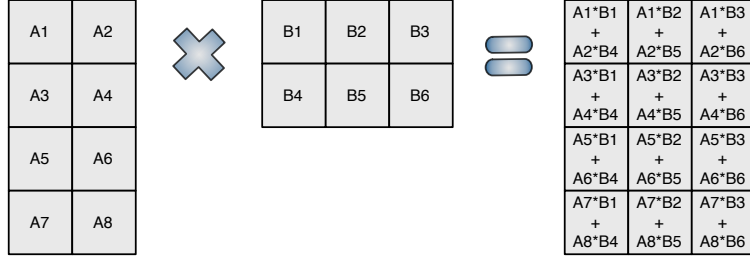


Figure 3: Block decomposition for the matrix multiplication problem

$$\varphi_s(\text{cacheLineSize}, \text{dist}, np) = \text{dist.getElementSize()} \times \lfloor \text{dist.getAveragePartitionSize}(np) + 0.5 \rfloor$$

where the result of `getAveragePartitionSize` is rounded-up to the closest integer to better suit the most common expected partition size.

A more conservative estimate ( $\varphi_c$ ) considers to some extent the two previously neglected dimensions, at the expense of more computational overhead:

$$\varphi_c(\text{cacheLineSize}, \text{dist}, np) = \text{cacheLineSize} \times \frac{\text{dist.getAveragePartitionSize}(np) \times \text{dist.getElementSize()}}{\text{dist.getAverageFirstDimSize}(np)} \times (\lceil \frac{\text{dist.getAverageFirstDimSize}(np)}{\text{cacheLineSize}} \rceil + 1)$$

Function `getAverageFirstDimSize` returns the average length (in number of elements) of the first dimension of the domain. This information is particularly important for the decomposition of multi-dimensional domains, specially multi-dimensional arrays<sup>2</sup>. We are assuming a row-major order memory layout and, in such context, the output of the `getAverageFirstDimSize` function is crucial to understand the breakdown of a partition into cache lines. The use of the average value conveys some extra information to the system when the size of the partitions is not uniform, as happens when the size in bytes of the sub-domains is not a multiple of  $np$ .

In  $\varphi_c$ , the size of the partition's first dimension is adjusted to the boundaries of the cache line. Furthermore, an extra cache line is added to consider the eventual misalignment of the partition to such boundaries. This approach is likely to ensure that the entire working set fits the TCL, but its conservative nature will eventually waste more space than the first approach. Table 2 illustrates, for both approaches, the estimated number of bytes that a partition of size  $\text{size}(p)$  will take in a cache with cache line size  $\text{size}(cl)$ .

As an example, consider the block decomposition for the parallel computation of the classic matrix multiplication problem illustrated in Figure 3. A partition of the domain must comprise a block of each input matrix and space for the computed result block, to be placed in the output matrix. Note, however, that every block partition of the first matrix ( $A$ ) must be paired with all the block partitions that compose a line of the second ( $B$ ). Consider now a concrete

<sup>2</sup>When targeting other kinds of data-structures, the default return value of `getAverageFirstDimSize` may simply be 1.

|                                |     | $\varphi_s$      |                      | $\varphi_c$   |   |
|--------------------------------|-----|------------------|----------------------|---|---|
|                                |     | Boundary-aligned | Not boundary-aligned | Boundary-aligned  | Not boundary-aligned  |
| Multiple of<br>cache line size | Yes | $size(p)$        | $size(p)$            | $size(p) + size(cl)$  | $size(p) + size(cl)$  |
|                                | No  | $size(p)$        | $size(p)$            | $size(cl) \times \frac{size(p)}{F} \times (\lceil \frac{F}{size(cl)} \rceil + 1)$ | $size(cl) \times \frac{size(p)}{F} \times (\lceil \frac{F}{size(cl)} \rceil + 1)$ |

Table 2: Estimated number of bytes that a multi-dimensional partition of size  $size(p)$  bytes (whose first dimension comprises  $F$  bytes) occupies in a cache with line size  $size(cl)$  bytes.

instance of the problem where both  $A$  and  $B$  are  $1024 \times 1024$  square matrices of 4-byte-long integers, and a TCL with 64 KBytes. For tentative  $np$  value  $256 = 16 \times 16$  the estimation given by  $\varphi_s$  is  $(\frac{1024}{16})^2 \times 3 \text{ matrices} \times 4 \text{ bytes} = 49152 \text{ bytes}$ , while the one given by  $\varphi_c$  is  $64 \times \frac{(\frac{1024}{16})^2 \times 3 \text{ matrices} \times 4 \text{ bytes}}{1024/16} \times (\lceil \frac{1024/16}{64} \rceil + 1) = 64 \times 64 \times 3 \text{ matrices} \times 4 \text{ bytes} \times (1 + 1) = 98304 \text{ bytes}$ . Thus  $np = 256$  is valid when using  $\varphi_s$  but not when using  $\varphi_c$ .

None of the presented  $\varphi$  functions take into consideration set associativity. The actual location of the data in the process' addressing space is not made available in many programming languages, e.g. Java. Moreover, the computational complexity required to take such knowledge into consideration would, most likely, subsume the eventual benefits. Nonetheless, considering a cache replacement algorithm of the *least-recently-used* family, the subjugation of a partition's size to the TCL's capacity highly contributes for having the delimited data fully loaded (minus conflicting cache lines) on such cache level.

## 2.2. Scheduling

The scheduling stage assigns pairs (instance of the computation, associated partition) - our *tasks* - to a set of workers. The problem diverges from the common scheduling of data-parallel computations because the amount of tasks generated by the cache-conscious decomposition approach largely exceeds the number of cores available in the machine.

Revisiting the application of the matrix multiplication algorithm of Figure 3 to matrices of dimension  $1024 \times 1024$ , each block of matrix  $A$  will have to be combined with 16 blocks of matrix  $B$ . Applying a one-to-one mapping from partitions to tasks will result in a total of  $16 \times 16 \times 16 = 4096$  tasks.

Spawning as many workers as tasks is not viable in this context, as having the number of execution flows far exceeding the number of computing resources penalizes performance. Also, given the small granularity of each task, to have a pure dynamic work-stealing-based scheduling policy will lead to considerable overheads, since the worker threads will spend a non-negligible percentage of their time fetching work, rather than executing it. Thus, performance in this context is highly dependent of an efficient and locality-aware mapping between tasks and workers. Our solution is to perform an initial static scheduling that assigns a cluster of tasks to each worker, which sequentially iterates upon the stream of the said tasks (recall Figure 2). We advocate that this static work distribution increases the system's overall performance, since workers do not have to search and compete for work. Nonetheless, when in the presence of irregular computations, dynamic scheduling techniques may also be useful to balance the load across the workers. We do not yet explore such techniques, even though we are aware of work in the field that already embeds some hierarchical concerns [10, 11].



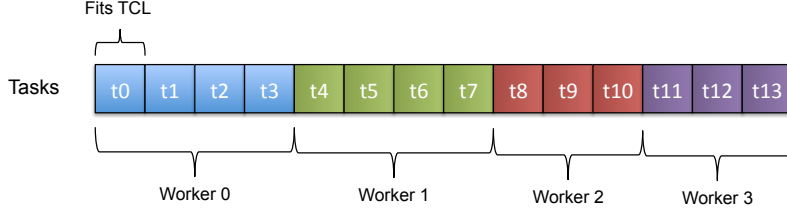


Figure 4: Contiguous Clustering; scheduling of 14 tasks among 4 workers

In the scope of this work we present two distinct task clustering strategies. The challenge is to trade-off the schedule’s efficiency against the overhead (temporal and spatial) that the determination of the next task to execute might impose on the overall execution. More complex clustering strategies are likely to perform more calculations and require more memory, thus stealing space in the cache for the actual computation’s data.

#### 2.2.1. Contiguous Clustering (CC)

Assigns an equally-sized contiguous cluster of tasks to each worker, according to their unique identifier. Given a set of  $n$  workers and a set of  $m$  tasks, a worker with rank  $i$  is assigned the following cluster of tasks:  $[(i \times \frac{m}{n}), ((i + 1) \times \frac{m}{n})]$ . Whenever  $m$  is not a multiple of  $n$ , the first  $r$  workers (for  $r = m \bmod n$ ) receive one extra task. Figure 4 illustrates the application of the CC strategy to the scheduling of 14 tasks among 4 workers.

The rationale behind this strategy is twofold: 1) introduce minimal overhead during scheduling, and 2) exploit spatial locality between tasks operating upon consecutive partitions.

#### 2.2.2. Sibling Round-Robin Clustering (SRRC)

Builds on the fact that the Last Level Cache (LLC) is shared by multiple computing cores. Thus, if two or more workers running in such cores share data, the number of LLC misses will decrement, reducing the accesses to main memory. Once again, a matrix multiplication example is paradigmatic, as multiple partitions share blocks of both input matrices.

The scheduling algorithm comprises two distinct assignment levels: the *cluster-assignment* level assigns clusters of tasks to groups of workers, whilst the *task-assignment* level assigns tasks within a cluster to workers within a worker group. In the first level, the size of the task clusters is ruled by the TCL to LLC ratio:

$$clusterSize = \frac{size(LLC)}{size(TCL)} + (cores(LLC) - (\frac{size(LLC)}{size(TCL)} \bmod cores(LLC)))$$

The second term simply ensures a proper distribution of the work when in the presence of remainder. It uses the number of cores that share a LLC, denoted by  $cores(LLC)$ , as the distribution unit.

Let we represent the resulting set of clusters ( $C$ ) as follows  $C = \{c_0, \dots, c_{n_c-1}\}$ , where  $c_i$  denotes a particular cluster, and  $n_c$  the number of clusters. Consider now that the set of workers ( $W$ ) may itself be grouped according to affinities of

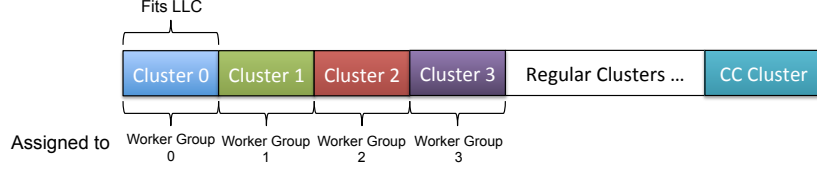


Figure 5: SRRC cluster-assignment: assignment of clusters of tasks among 4 groups of workers running on sibling cores that share a LLC.

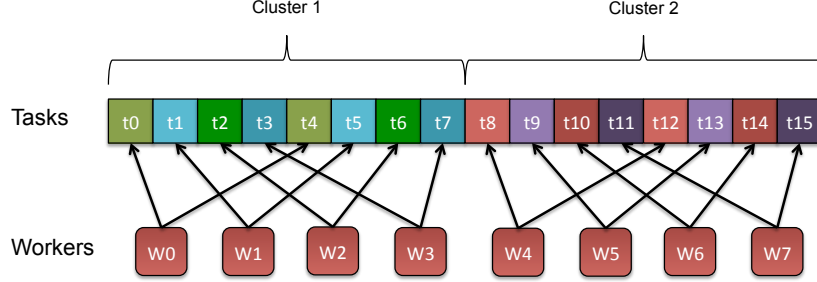


Figure 6: SRRC task-assignment: assignment of the tasks within a cluster among the workers that compose the target group.

the cores (where they will carry out their execution) to the LLC. Accordingly,  $W$  may be represented as  $W = \{w_0, \dots, w_{n_w-1}\}$  where  $w_i$  denotes a particular worker-group and  $n_w$  denotes the number of such groups. Given these definitions, the scheduling of  $C$  among  $W$  follows a round-robin strategy that assigns, to each group of workers  $w_i \in W$ , the following subset of clusters:

$$C_i = \{c_j \in C \mid j \bmod n_w = i \wedge j < (n_c - (n_c \bmod n_w))\}$$

To guarantee a schedule as even as possible, when the number of clusters is not a multiple of work groups, the remainder clusters are merged in a special cluster, named *CC Cluster*. This cluster also comprises the tasks that could not form a cluster (given by  $clusterSize \times (n_t \bmod (clusterSize \times n_w))$ ) and is scheduled according to the CC strategy.

Figure 5 illustrates the cluster-assignment for a machine with 4 groups of sibling cores sharing 4 different LLCs, along with the assignment of the resulting clusters to the worker groups. The task-assignment within a cluster (illustrated in Figure 6) is performed in a *round-robin* fashion among the workers that compose the group.

### 2.3. Worker-Core Affinity

The thread scheduling policies of modern operating system are aware of the threads' cache footprints, keeping threads on the same core as much as possible. Nonetheless, in some cases we need to compulsorily constrict the set of cores on which a thread may execute, namely to properly apply the SRRC strategy. This strategy assumes that the workers operating over a given task cluster execute on cores sharing a LLC. Therefore, it is important to map the affinity between workers and cores accordingly.

In order not to be too restrictive, we allow worker threads to be freely scheduled among the cores under the lowest shared cache level, a strategy we have adequately baptized as *Lowest Level Shared Cache* affinity mapping. As an example, in a quad-core architecture with dedicated L1 caches, a L2 cache shared by each two cores, and a single L3 cache, the Lowest-Level-Shared-Cache affinity mapping allows the operating system to freely schedule worker threads between every two cores that share a L2 cache.

#### 2.4. Synchronization-free Execution Engine

To leverage the devised cache-conscious domain decomposition strategy, the underlying execution engine must provide efficient access from each worker thread to the tasks assigned to it by the scheduling policy in place. Our approach is to allow the workers to directly access the tasks generated by decomposition stage, which are stored contiguously in a vector. This avoids the performance penalty of moving tasks to worker-local data-structures. Accordingly, each thread iterates through the shared vector to sequentially fetch and execute each task scheduled to it.

All accesses to the shared task vector are synchronization-free. This is possible because each worker receives a disjoint set of task clusters, and the associated index sets are locally computable. From its unique rank identifier, a worker is able to determine the index of the first task to execute, and the relative position of all the remainder. The computational complexity of this operation is bound to the scheduling policy. The SRRC approach requires two loops to iterate over the whole set of tasks assigned to it and needs to deal with remainders within and across clusters. In turn, the CC counterpart requires a single loop over a contiguous vector.

### 3. Implementation Details

We have prototyped our proposal in the Elina Java parallel computing framework [12, 13] for distributed and shared-memory environments. Elina supports both embarrassingly parallel data-parallel and MapReduce computations. Moreover, it is a very modular framework that cleanly separates most of the system-level functionalities into independent modules, whose concrete implementation is specified via a configuration file. As a result, equipping the framework with a new implementation of given module, such as the decomposition or scheduling strategy, requires *only* the implementation of a pre-defined interface, and an altering a configuration file. This option allowed us to, on one hand, have programming and execution models close to what may be found in the most used MapReduce-based frameworks for the processing of large datasets, such as Hadoop [14], and on the other hand, to evaluate multiple system solutions by simply modifying the framework’s configuration, without having to modify the framework’s core or the application’s source code.

Our implementation efforts were thus directed to the development of multiple adapter modules, namely for the domain decomposition strategy (cache-conscious and horizontal), the scheduling algorithm (*Contiguous Clustering* and *Sibling Round-Robin Clustering*), the *Lowest Level Shared Cache* affinity mapping, among others. Here we will just briefly review how we uniformly represent memory hierarchies in the framework, and how we have implemented the thread-to-cache affinities.

---

```

{
  "siblings": [[0,2,4,6],[1,3,5,7]],
  "size": 4294967296,
  "child": {
    "siblings": [[0,2,4,6],[1,3,5,7]],
    "size": 6291456,
    "cacheLineSize": 64,
    "child": {
      "siblings": [[0],[1],[2],[3],[4],[5],[6],[7]],
      "size": 524288,
      "cacheLineSize": 64,
      "child": {
        "siblings": [[0],[1],[2],[3],[4],[5],[6],[7]],
        "size": 65536,
        "cacheLineSize": 64,
        "child": null
      }
    }
  }
}

```

---

Listing 1: A NUMA node comprising two quad-core CPUs and 8 GBytes of RAM - 4 Gbytes per CPU. Each CPU features a single L3 cache and four L1 and L2 caches - one per core.

### 3.1. Platform-Independent Representation of a Memory Hierarchy

There is no standard format for the storing of hardware related information by operating systems, nor there is a single tool that provides such information for (at least) the most known operating systems, such as Windows, Linux and Mac OS X. Accordingly, we had to develop our own platform-independent representation of a memory hierarchy. For that purpose we use the JSON data representation format. A memory hierarchy representation is hence defined as a set of nested JSON objects comprising the following fields:

**size** - size of each individual memory element that composes the current memory level (in bytes), such as the RAM memory associated to a given CPU or the size of a particular cache memory.

**cacheLineSize** - size of the cache coherency line (in bytes). This field is present only if the memory level represents a cache level.

**siblings** - array of arrays of sibling cores sharing each copy of the memory level.

**child** - object containing the memory level information of the lower (child) level (is null if the current level is the bottom-most in the hierarchy).

As a proof of concept we implemented a tool that automatically generates a platform-independent JSON representation of a node's memory hierarchy from the information stored in the `/sys/devices/system/cpu/` directory of a Linux installation. Listing 1 showcases the result obtained for a NUMA (Non-Uniform Memory Access) architecture comprising two quad-core CPUs and 8 GBytes of RAM.

### 3.2. Thread to Cache Affinities in Java

The Java standard API does not provide means for establishing affinities between threads running in the Java virtual machine and cores of the underlying processor(s). Ergo, to implement the *Lowest Level Shared Cache* affinity

mapping we had to resort to external commands, namely `jstack` to obtain the correspondence between the Java virtual machine’s and the operating system’s thread identification, and `taskset` to set the affinity of the threads according to the *Lowest Level Shared Cache* policy. Note that while the `jstack` tool is cross-platform, `taskset` is only available in (most) Linux distributions.

## 4. Evaluation

Our experimental evaluation aims to characterize which kind of applications may benefit from the devised cache-conscious decomposition approach, and to quantify such benefit. For that purpose, we carried out a comparative performance analysis against the pre-existing horizontal work distribution featured in Elina. Elina’s portable programming model and run-time system allowed us to use the same source code on both settings. We simply deployed the run-time system with different instances of several modules, namely of the decomposition strategy, the scheduling strategy, and the  $\varphi$  function.

### 4.1. Test Infrastructure

All experiments were conducted on two different types of nodes, with the following characteristics:

**System A** - 2 Quad-Core AMD Opteron™ Processor 2376 with three cache levels: a 64KBytes L1 data cache per core, a unified 512KBytes L2 cache per core, and a unified 6MBytes L3 cache per processor.

**System I** - 2 Dual-Core Intel(R) Xeon(R) CPU X30 hyperthreaded with three cache levels: a 32KBytes L1 data cache per core, a unified 256KBytes L2 cache per core, and a unified 8MBytes L3 cache per processor.

All nodes run the Debian Linux distribution with Linux kernel version 2.6.26-2-amd64. The installed Java platform is OpenJDK 7 (version 1.7.0.21).

### 4.2. Benchmarks

To conduct our study we chose seven benchmarks. The first two are widely used operations over matrices, namely matrix multiplication (**MatMult**) and matrix transpose (**MatTrans**). The problem class indicator for both benchmarks represents the dimension of the matrices involved (only square matrices were considered).

Gaussian Blur (**GaussianBlur**) blurs an image (represented as a matrix) by convolving it with a Gaussian function. The benchmark requires two parameters: the image to blur and the radius of the blurring window for each pixel. The problem class indicator follows notation  $S$ - $R$ , where  $S$  stands for the target image’s dimension and  $R$  for the blurring radius.

**Crypt**, **SOR** and **Series** are adapted from the JavaGrande benchmark suite. Crypt was adapted to cipher and decipher files instead of messages. The problem class indicator is bound to the size of the file to cipher/decipher. SOR computes the Successive Over Relaxation algorithm for a matrix of dimension  $N \times N$ , whilst Series computes the first  $N$  Fourier coefficients of the function  $f(x) = (x + 1)^x$  on the interval  $[0, 2]$ . On both these benchmarks, the problem class indicator denotes the value of parameter  $N$ .

---

```

1  public class IntArray2DDistribution extends AbstractDistribution<int[]> {
3      private final int numColumns;
4      private final int numRows;
6      ... // constructors
8      public int[][] partition(int np) {
9          // partition a matrix into np blocks
10     }
12     public int validate(int np) {
13         float sqrt = Math.sqrt(np);
14         float rsqrt = Math.round(sqrt); // may be cached for performance reasons
15         return (sqrt == rsqrt) ? 1 : 0;
16     }
18     public int getIndivisibleSize(int np) {
19         return 1;
20     }
22     public float getAveragePartitionSize(int np) {
23         float rsqrt = Math.round(Math.sqrt(np)); // may read value from cache
24         return (numColumns * numRows)/(rsqrt*rsqrt);
25     }
27     public float getAverageFirstDimSize(int np) {
28         float rsqrt = Math.round(Math.sqrt(np)); // may read value from cache
29         return numColumns/rsqrt;
30     }
31 }

```

---

Listing 2: Distribution algorithm for the cache-conscious decomposition of two-dimensional arrays

Finally, **WordCount** is an implementation of the classic word-count MapReduce example. As in Crypt, the problem class indicator is bound to the size of the file to process.

#### 4.3. Implementing the Distribution Interface

We argue that the effort that our approach demands from the programmer is relatively small, when compared with the potential performance gains - a claim that cannot be sustained by systems that promote the explicit programming of the memory hierarchy, such as Sequoia [2], due to the programming labour involved.

To justify our allegation, we present in Listing 2 a concrete implementation of the `Distribution` interface for the decomposition of two-dimensional integer arrays. The focus is on the modifications imposed on the pre-existent distribution algorithm, in order for it to comply to the new interface<sup>3</sup>. For that reason we omit the implementation of method `partition`, given that it is independent of the decomposition strategy. Method `getElementSize` is also absent but only for code reuse reasons. The implementation is inherited from the base class, and simply returns the size (in bytes) of an element of the array: 4 in this case. The implementation of the remainder methods is fully depicted in the listing.

---

<sup>3</sup>This compliance is a necessary and sufficient condition for the application of the cache-conscious decomposition strategy.

| Benchmark             | Class   | Speed-up VS Horizontal |          | Speed-up VS Sequential |          |
|-----------------------|---------|------------------------|----------|------------------------|----------|
|                       |         | System A               | System I | System A               | System I |
| Matrix Multiplication | 1000    | 1,21                   | 3,36     | 33,42                  | 14,69    |
|                       | 1500    | 7,46                   | 5,26     | 46,52                  | 17,96    |
|                       | 2000    | 7,22                   | 6,51     | 49,96                  | 20,68    |
| Matrix Transpose      | 3500    | 4,27                   | 4,17     | 30,30                  | 14,29    |
|                       | 5000    | 4,63                   | 5,11     | 37,92                  | 17,28    |
|                       | 10000   | 5,64                   | 6,40     | 48,13                  | 24,20    |
| Gaussian Blur         | 1000-15 | 2,54                   | 2,18     | 7,61                   | 4,50     |
|                       | 1000-20 | 2,82                   | 2,29     | 7,42                   | 4,47     |
|                       | 1000-25 | 3,06                   | 2,57     | 7,29                   | 4,26     |
| SOR                   | 2000    | 3,19                   | 2,77     | 10,99                  | 9,28     |
|                       | 4000    | 3,49                   | 2,89     | 11,11                  | 9,27     |
|                       | 10000   | 3,70                   | 3,17     | 11,97                  | 8,77     |

Table 3: Cache-conscious versus horizontal decomposition and versus the sequential version - Temporal locality sensitive benchmarks

| Benchmark  | Class    | Speed-up VS Horizontal |          | Speed-up VS Sequential |          |
|------------|----------|------------------------|----------|------------------------|----------|
|            |          | System A               | System I | System A               | System I |
| Crypt      | 9.5 MB   | 1,00                   | 1,00     | 7,33                   | 6,05     |
|            | 95.5 MB  | 1,00                   | 1,00     | 7,58                   | 6,01     |
|            | 190.7 MB | 0,99                   | 1,00     | 7,80                   | 6,15     |
| Series     | 10000    | 0,99                   | 0,99     | 7,99                   | 6,77     |
|            | 100000   | 1,00                   | 1,01     | 7,30                   | 6,81     |
|            | 1000000  | 1,00                   | 1,00     | 7,11                   | 7,27     |
| Word Count | 5,3 MB   | 0,99                   | 0,99     | 6,20                   | 5,83     |
|            | 74,3 MB  | 0,99                   | 0,99     | 6,16                   | 6,41     |
|            | 297.0 MB | 1,01                   | 1,01     | 6,23                   | 6,16     |

Table 4: Cache-conscious versus horizontal decomposition and versus the sequential version - No temporal locality sensitive benchmarks

Note that the distribution forces the array to be partitioned into as many blocks per column as for row, hence why the `validate` method forces  $np$  to be a perfect square.

#### 4.4. Performance Evaluation

##### 4.4.1. Cache-conscious versus horizontal decomposition

Tables 3 and 4 depict the performance gains relatively to the canonical horizontal decomposition and a sequential version of the benchmarks. A color scale allows to quickly identify the best and worst case scenarios. Darker the background color, better is the result. The comparison with the sequential version of the benchmarks are presented only to a given a better perceptive of the overall gains delivered by either approach.

The benchmarks can be clearly classified into two groups: on Table 3 are the ones where cache-conscious decomposition brings considerable performance gains over horizontal decomposition, and on Table 4, the ones where both decomposition strategies are on a par. These results are clearly bound to the cache-locality properties of the benchmarks. As could be expected, only the ones featuring both spatial and temporal locality - MatMult, MatTrans, GaussianBlur and SOR - really benefit from the cache-conscious decomposition. The charts in Figures 7 and 8 graphically substantiate this statement, presenting the speed-up obtained by both decomposition approaches in the two systems. The

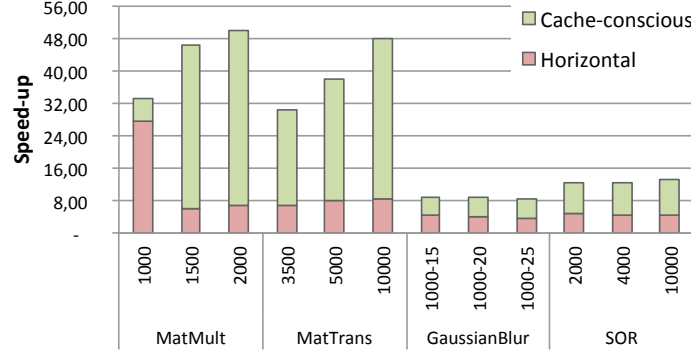


Figure 7: Speed-up of both decomposition approaches vs the sequential implementation in System A

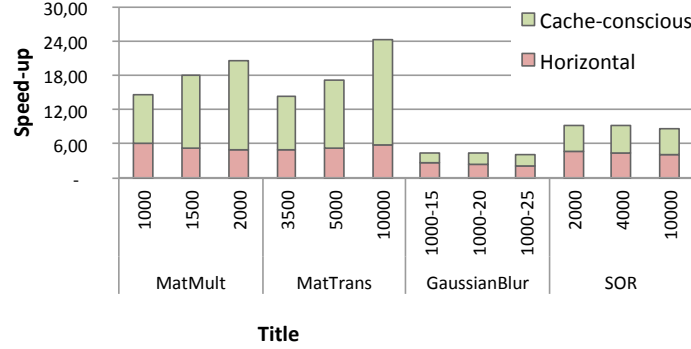


Figure 8: Speed-up of both decomposition approaches vs the sequential implementation in System I

results are better for System A than for System I because, in the former, each core runs a single hardware thread, while, in the latter, cores are *hyperthreaded* and thus run two hardware threads. Consequently, in System A, caches are shared by less threads and hence there are less conflict and capacity misses.

The performance gains for MatMult and MatTrans are considerable (up to 6 - 7 times), specially as the size of the problem increases. A major performance boost, since we are in presence of a data locality optimization not extra parallelization. The results obtained for MatMult 1000 in System A escape the norm, thus deserving a more careful explanation. Given the size of the matrix, the number of workers, and the size of the L1 cache (64 KBytes), the horizontal decomposition produces partitions not much bigger than the L1 cache. Consequently, for this particular parameterization, the computation is already cache-friendly and there is not much room for cache-related optimizations. The same does not apply to System I because the cache is smaller (32 Kbytes) and is shared by two hardware threads.

The gains for GaussianBlur and SOR are not as impressionable, but still very good: up to approximately 3 times better than the horizontal decomposition. Note that these two benchmarks are stencil computations that, by operating over 2-dimensional neighbourhoods, already exhibit data access patterns that



|              |         | SRRC scheduling |                |                |          |                |                | CC scheduling |                |                |                  |          |                |                |                  |
|--------------|---------|-----------------|----------------|----------------|----------|----------------|----------------|---------------|----------------|----------------|------------------|----------|----------------|----------------|------------------|
|              |         | System A        |                |                | System I |                |                | System A      |                |                |                  | System I |                |                |                  |
| Benchmark    | Class   | TCL             | Speed-up vs Hz | Speed-up vs L1 | TCL      | Speed-up vs Hz | Speed-up vs L1 | TCL           | Speed-up vs Hz | Speed-up vs L1 | Speed-up vs SRCC | TCL      | Speed-up vs Hz | Speed-up vs L1 | Speed-up vs SRCC |
| MatMult      | 1000    | 128k            | 1,22           | 1,04           | 64k      | 3,40           | 1,09           | 128k          | 1,21           | 1,04           | 0,99             | 64k      | 3,36           | 1,09           | 0,99             |
|              | 1500    | 128k            | 7,71           | 1,11           | 64k      | 5,34           | 1,09           | 128k          | 7,46           | 1,00           | 0,97             | 64k      | 5,26           | 1,09           | 0,98             |
|              | 2000    | 128k            | 7,20           | 1,00           | 64k      | 6,54           | 1,09           | 128k          | 7,22           | 1,02           | 1,00             | 64k      | 6,51           | 1,11           | 1,00             |
| MatTrans     | 3500    | 192k            | 4,00           | 1,07           | 64k      | 4,55           | 1,40           | 256k          | 4,27           | 1,03           | 1,07             | 64k      | 4,17           | 1,12           | 0,91             |
|              | 5000    | 192k            | 4,27           | 1,09           | 128k     | 5,28           | 1,21           | 192k          | 4,63           | 1,07           | 1,08             | 128k     | 5,11           | 1,17           | 0,97             |
|              | 10000   | 192k            | 5,27           | 1,15           | 128k     | 6,20           | 1,12           | 256k          | 5,64           | 1,11           | 1,07             | 64k      | 6,40           | 1,13           | 1,03             |
|              | 1000-15 | 128k            | 2,54           | 1,13           | 192k*    | 2,18           | 1,00           | 192k          | 2,41           | 1,19           | 0,95             | 192k*    | 2,12           | 1,00           | 0,97             |
| GaussianBlur | 1000-20 | 128k            | 2,82           | 1,00           | 192k*    | 2,29           | 1,01           | 192k          | 2,63           | 1,04           | 0,93             | 192k*    | 2,22           | 1,01           | 0,97             |
|              | 1000-25 | 128k            | 3,06           | 1,00           | 128k*    | 2,57           | 1,00           | 128k          | 2,81           | 1,01           | 0,92             | 128k*    | 2,49           | 1,00           | 0,97             |

Table 5: Speed-ups of the CC and SRRC strategies varying the TCL size: summary table. The TCL values bearing the \* mark are the smallest size for which valid partitioning solutions were found.

leverages cache to some extension. Naturally, as the size of the matrices and/or size the neighbourhood increases, this property fades and the gains of cache-conscious decomposition are more noticeable. Also note that in GaussianBlur the computation is somehow unbalanced due to the processing of the image’s borders. This fact can be further observed in the speed-ups against the sequential version, which are relatively lower when compared with the remainder benchmarks. This is true for both decomposition strategies.

The benchmarks in the second set do not benefit from temporal locality, and are thus ideal for determining the overhead imposed by the cache-conscious decomposition, namely by the generation of a large amount of tasks and their subsequent scheduling. In the tested benchmarks no significant performance gains or losses can be found. Crypt and Series iterate data sequentially (benefiting from spatial locality), without revisiting previously accessed data. Hence, no benefits are attained from enforcing temporal locality by keeping the partitions’ size within the TCL. Conversely, WordCount features temporal locality on the access to the map that stores the number of word occurrences. However, the random access pattern to such data precludes any attempt to, in advance, predict which data to place in the TCL. A cache-aware implementation of such map is a challenge to overcome. In sum, the presented results show that the overhead of our approach is negligible, attesting that it may be used in a wide range of applications, without concerns for the type of locality.

#### 4.4.2. Sensitivity to the chosen TCL

Next we perform a sensitivity analysis of the results relatively to the chosen TCL and its size. To that end we vary the size of the TCL from the L1 to the L3 cache sizes. Given that the results are similar across all systems and scheduling strategies, we limit our discussion to just one configuration: CC scheduling strategy in System A (Figure 9), and present the essence of the analysis in Table 5. The results are very insightful, as the optimal TCL value lies somewhere between the sizes of the L1 and the L2 caches and is benchmark and architecture dependent. We associate this to the fact that: (a) the JVM itself has a state that competes for space in the cache, and (b) we are not considering all the dimensions of cache hardware, particularly the number of ways, and hence are permeable to conflict misses.

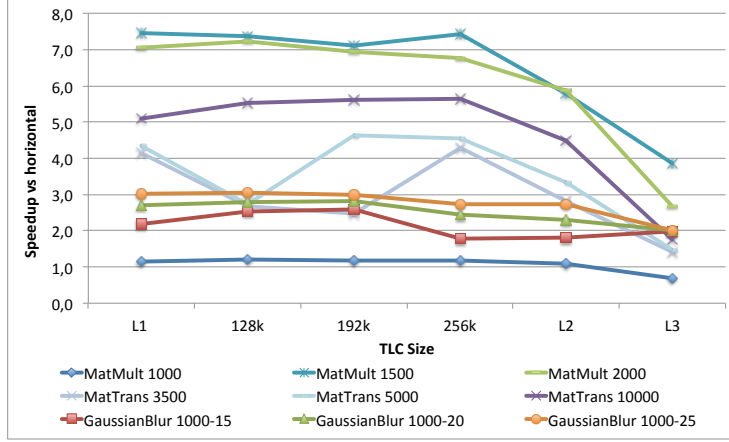


Figure 9: Speed-ups of the CC strategy in System A, varying the size of the TCL

#### 4.4.3. Sensitivity to the scheduling strategies and to the $\varphi$ function

Also from Table 5 it can be observed that, for most cases, the impact of employing the CC or the SRRC scheduling strategies on the benchmark’s absolute performance is not substantial. The TCL for which the benchmark attains its performance peak does depend on the scheduling strategy employed, but the absolute performance of such peaks are very close - the essential gains result from the base cache-conscious decomposition strategy. GaussianBlur is the sole benchmark to consistently benefit from one of the strategies, SRRC, for higher radius values: larger blur window increases the amount of data that is shared by tasks operating over contiguous windows.

Regarding the  $\varphi$  functions, we also evaluated the benchmarks using the two  $\varphi$  functions presented in Section 2.1. The results showed that cache-line-awareness did not improve the performance for any of the assessed cache sizes. In fact, the results were worse, given the introduced overhead and the wasted cache space of the conservative approach.

#### 4.4.4. Breakdown

In order to better evaluate the overhead imposed by the proposed cache-conscious approach (denoted *CacheCons* in Figure 10), we broke down the execution of the MatMult benchmark, in system A, for  $N=2000$  and TCL size of 128k (the best performance). **Decomposition** and **Scheduling** denote the time spent in the decomposition of the domain and the subsequent scheduling of the tasks, **Execution** denotes the time spent in the actual matrix multiplication, and **Reduction** represents the time spent in the reduction of the partial results (see Figure 3).

As depicted in the figure, the weight of the stages other than **Execution** is one order of magnitude higher in the cache-conscious than in the horizontal approach. This impact is mostly visible in the **Reduction** stage, that in the cache-conscious cases reaches almost 5% of the whole execution time, given that that number of results to reduce at node-level is much higher. The remainder **Decomposition** and **Scheduling** stages are fully optimized for either clustering

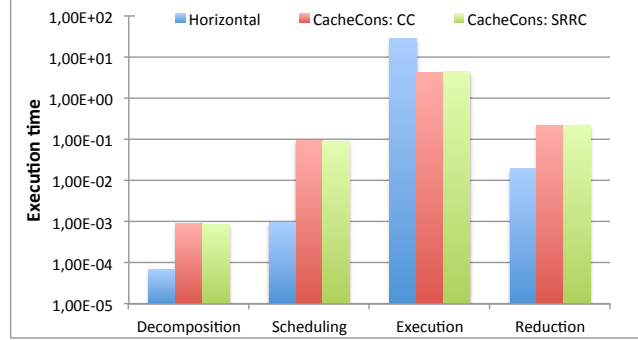


Figure 10: Breakdown of MatMult N=2000 (System A) - logarithmic scale

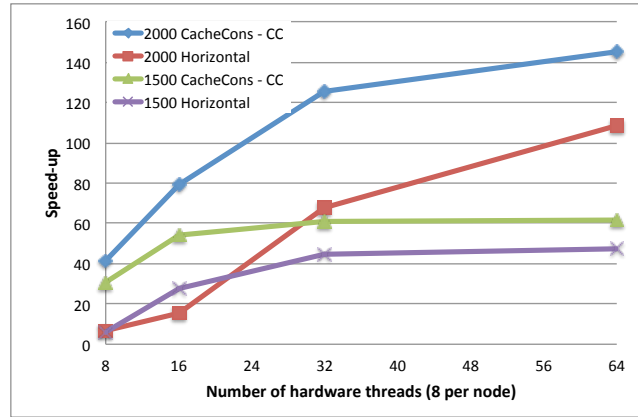


Figure 11: Speed-up of the MatMult benchmark - Cluster environment

strategy, pertaining to less than 2% of the total execution time. In this particular example, the cache-conscious decomposition generates 8000 tasks (1000 per each of the 8 workers), that are created and scheduled in less than a 0.1 seconds.

Despite the overhead imposed in the three aforesaid stages, the overall performance of the cache-conscious is much better. This is due to the substantial gains obtained in the heavier **Execution** stage, that takes more than 90% of the total execution time in all three cases.

#### 4.5. Impact at Cluster-Level

To illustrate the impact of our approach at cluster-level, we present the speed-up of two instances of the MatMult benchmark relatively to the sequential execution, when varying the number of nodes: 1, 2, 4 and 8 (each featuring 8 hardware threads). Figure 11 depicts the curves for the horizontal and cache-conscious decompositions. For the 2000 problem class, both approaches scale up to 8 nodes (64 hardware threads). Naturally, the node-level gains of the cache-conscious approach transpire in the overall execution time. The performance peak of the cache-conscious decomposition delivers a 137 speed-up over the sequential execution, and a 3.7 speed-up over the horizontal approach. In the horizontal approach, as the number of nodes increases, the data partition

assigned to each node, and subsequently to each worker thread, diminishes and approximates itself to the size of the higher cache levels. This performance increase is most noticeable from the 2 to the 4 node configuration. As such, one can also leverage the impact of cache locality in horizontal decompositions. However, these gains are ephemeral and do not scale, giving that they are bound to the assignment of small data partitions to nodes. Conversely, cache-conscious decomposition delivers higher gains for data-sets of big dimensions. In this sense, it is particularly suitable for cluster environments, where the scale of the problems to solve is, by definition, large.

## 5. Related Work

### 5.1. Hierarchical Data Parallelism

Our work is close to the hierarchical data parallelism field. However, contrary to our proposal, hierarchical data parallelism models place on the programmer’s shoulders the burden of decomposing the domain according to the traits of the memory hierarchy. Of these models, Sequoia [2, 15, 16] is the most prominent. It provides a programming language for the explicit programming of the memory hierarchy. The language’s main program building block is the task, a side-effect free function with call-by-value parameter passing semantics. Through tasks the programmer is able to express: parallelism, explicit communication and locality, isolation, algorithmic variants and parameterization. These properties allow programs written in Sequoia to be portable across machines without sacrificing the possibility of tuning the application for each one. Tasks run in isolation, and hence may be executed concurrently without requiring synchronization between cooperating threads. Parameter passing during task launching is the only communication mechanism available, which increases the complexity of expressing cooperative computations. Hierarchical awareness is achieved by providing different implementations (variants) of a given task. *Inner* variants reflect intermediate nodes in the memory hierarchy and have the purpose of decomposing the input dataset. *leaf* variants perform the actual computation, operating directly on working sets residing within leaf levels of the memory hierarchy. Mapping a hierarchy of tasks onto the hierarchical representation of memory requires the creation of task instances for every machine level involved. The programmer is required to provide the compiler with the task mapping specification for the machine where the algorithm will be compiled and executed.

Parallelism in Sequoia is assumed to be regular. Additional constructs to support irregular parallelism are proposed in [16], namely the *call-up* and *spawn*. *call-up* allows subtasks to access their parent’s heap, which can be used to modify its data structures. *spawn* provides for the dynamic generation of parallelism, being able to launch an arbitrary number of subtasks of the provided task.

Hierarchically Tiled Arrays (HTA) [4] is a programming paradigm that relies on an object type named *tiled array* for expressing parallelism and locality. HTAs are arrays partitioned into tiles, which can in turn be either conventional arrays or further tiled arrays, enabling hierarchical decomposition. The components of an HTA can be accessed in a way analogous to the conventional array indexing. Once more, the hierarchical decomposition is explicit in the program: the *level* function can be used to obtain, at runtime, the location of the argument within the tile hierarchy.

The Hierarchical Place Trees (HPT) [3] model combines concepts from Sequoia and the X10 languages. From the latter it borrows the concepts of **place** and **activity** (task). It abstracts each memory module as a place, and therefore a memory hierarchy is abstracted as a *place tree*. Places are tagged with annotations that indicate their memory type and size. Moreover, a processor core is abstracted as a *worker thread* which in the HPT model, can only be attached to leaf nodes in the place tree. In contrast with Sequoia, HPT supports three different types of communication: implicit access, explicit in-out parameters, and explicit asynchronous transfer.

Less noticeable works are that also address hierarchical decomposition are [6] and [7]. These systems perform horizontal decompositions at both the cluster and the node level, firstly decomposing the domain at hand by the multiple nodes that compose the distributed environment, and secondly by the pool of workers available at each node.

There are several fundamental differences between our work and hierarchical data parallelism. The most evident is (as previously stated) that we do propose a systematic approach to the memory-hierarchy-aware decomposition of data-sets, and not programming abstractions for the programmer to do so. We do ask for some help from the programmer in the implementation of the **Distribution** interface, but it is architecture agnostic information. Moreover, none of these models can be applied to the class of computations that we are aiming for: computations where the decomposition stage is cleanly separated from the computation stage. Hierarchical data parallel models follow a divide-and-conquer approach that blends the two stages together. Finally, in rigor, we do not follow a hierarchical approach, as we do not iteratively partition a domain so that it first fits the higher memory level, and from then on, each of the inferior levels. We delegate such enterprise of the scheduling strategy. For instance, the SRCC strategy tries to keep in each LLC the stream of partitions that will be feed to workers running in the cores bound to that cache level.

### 5.2. In-Memory MapReduce

There has been quite some work on the optimization of the MapReduce execution model to the intra-node reality. These efforts, commonly referred to *in-memory MapReduce* show some concern about data locality. For instance, Phoenix [17] adjusts the size of the input and output data of a *map* task, so that this data can fit in the L1 cache, which reveals a concern about the utilization of the cache. However, no attention is paid to the layout of the data in memory, nor to the overall organization of the memory hierarchy, namely the core-to-cache affinities and the degree of sharing of cache levels among cores. It is just an optional user-tuned parameter that conveys information about the preferred size for each partition.

Another system that presents a rationale close to ours is Tiled-MapReduce [18]. It employs a pipeline of *map* and *reduce* tasks that make use of the same memory spaces and reduces idle time of the processing units, promoting locality. Furthermore, it also makes use of tiling strategies for the partition of the domain. However, this tiling process does not take the memory hierarchy into consideration, a key contribution of our work.

Other data locality related concerns have been addressed in in-memory MapReduce implementations. Phoenix++ [19] and Metis [20] have directed their focus to the efficient implementation of the data structures that harbor

the intermediate results of the *map* stage. Additionally, Phoenix++ also tries to take advantage of locality by applying the *combiner* stage every time a new intermediate result is emitted. The motivation is to, in an ad hoc manner, benefit from the likely possibility of the result still residing in a lower level cache.

In turn, HJ-Hadoop [21] extends Hadoop with features of the Habanero Java framework for multi-core parallelism. Data locality is explored by reducing the number of Java virtual machines created per node, allowing for a more efficient parallel execution of the *map* stage and for the buffering of the data feed to the user-defined functions.

## 6. Conclusions

In this paper we defined and implemented a systematic cache-conscious strategy for decomposing the domain of an application according to the traits of the target machine’s memory hierarchy. A performance evaluation demonstrates the advantage of the approach when targeting computations that feature temporal locality in the access to data. We have obtained up to a 7.7 speed-up relatively to the standard horizontal decomposition in this class of computations. In the remainder cases, no performance penalties were observed, foreseeing a wide applicability of the solution.

Another important conclusion drawn from this work is that the best clustering strategy, and TCL size configuration, is computation and architecture-dependent. It is not possible to systematically use the same execution settings across applications and architecture, which compromises performance portability. To mitigate this problem, we are currently addressing the automatic inference of these configurations. The goal is to design and implement an auto-learning stage that, over time, progressively learns the best configurations to be applied for each problem and its input sizes, applying these settings upon a request to execute the given problem.

Finally, we have also presented initial evidences that cache-conscious decomposition is also of particular usefulness in cluster environments, as it improves the data locality of algorithms that manipulate large data-sets. Future work will assess its applicability to the area of Big Data analytics.

## References

- [1] K. S. McKinley, S. Carr, C.-W. Tseng, Improving data locality with loop transformations, *ACM Trans. Program. Lang. Syst.* 18 (4) (1996) 424–453.
- [2] K. Fatahalian, et al., Sequoia: programming the memory hierarchy, in: *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, ACM Press, 2006, p. 83.
- [3] Y. Yan, et al., Hierarchical place trees: a portable abstraction for task parallelism and data movement, in: *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing (LCPC’09)*, Springer-Verlag, 2010, pp. 172–187.
- [4] G. Biksh, et al., Programming for parallelism and locality with hierarchically tiled arrays, in: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006)*, 2006, pp. 48–57.

- [5] S. Treichler, M. Bauer, A. Aiken, Language support for dynamic, hierarchical data partitioning, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, ACM, 2013, pp. 495–514.
- [6] L. Wang, S. Merchant, T. El-Ghazawi, Exploiting hierarchical parallelism using UPC, in: Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPS Workshops '11), IEEE Computer Society, 2011, pp. 1216–1224.
- [7] A. Kamil, K. Yelick, Hierarchical computation in the SPMD programming model, in: Proceedings of the 26th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2013), Vol. 8664 of LNCS, Springer, 2014, pp. 3–19.
- [8] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, *Commun. ACM* 51 (1) (2008) 107–113.
- [9] M. Frigo, et al., Cache-oblivious algorithms, in: 40th Annual Symposium on Foundations of Computer Science, FOCS '99, IEEE Computer Society, 1999, pp. 285–298.
- [10] J.-N. Quintin, F. Wagner, Hierarchical work-stealing, in: Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Vol. 6271 of Lecture Notes in Computer Science, Springer, 2010, pp. 217–229.
- [11] G. Zheng, et al., Hierarchical load balancing for Charm++ applications on large supercomputers, in: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPP' 10 Workshops), IEEE Computer Society, 2010, pp. 436–444.
- [12] J. Saramago, D. Mourão, H. Paulino, Towards an adaptable middleware for parallel computing in heterogeneous environments, in: 2012 IEEE International Conference on Cluster Computing Workshops, CLUSTER Workshops 2012, IEEE, 2012, pp. 143–151.
- [13] H. Paulino, E. Marques, Heterogeneous programming with Single Operation Multiple Data, *J. Comput. Syst. Sci.* 81 (1) (2015) 16–37.
- [14] Apache Inc., Apache hadoop, <http://hadoop.apache.org/> (2015).
- [15] M. Houston, et al., A portable runtime interface for multi-level memory hierarchies, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, ACM, 2008, pp. 143–152.
- [16] M. Bauer, J. Clark, E. Schkufza, A. Aiken, Programming the memory hierarchy revisited: supporting irregular parallelism in Sequoia, in: Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, ACM, 2011, pp. 13–24.

- [17] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, C. Kozyrakis, Evaluating mapreduce for multi-core and multiprocessor systems, in: 13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), ACM, 2007, pp. 13–24.
- [18] R. Chen, H. Chen, Tiled-mapreduce: Efficient and flexible mapreduce processing on multicore with tiling, TACO 10 (1) (2013) 3.
- [19] J. Talbot, R. M. Yoo, C. Kozyrakis, Phoenix++: modular mapreduce for shared-memory systems, in: Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11), ACM, 2011, pp. 9–16.
- [20] Y. Mao, R. Morris, M. F. Kaashoek, Optimizing mapreduce for multicore architectures, Tech. rep., Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology (2010).
- [21] Y. Zhang, Hj-hadoop: an optimized mapreduce runtime for multi-core systems, in: Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '13 - Companion Volume, ACM, 2013, pp. 111–112.